Tutorial: Representation of an Opening Book Tree

Kathe Spracklen

Fidelity Electronics
Miami, FL

An opening book is a collection of grandmaster moves for the start of play in a chess game. Advantages that result from the inclusion of an opening book are that the pieces can be developed in a coordinated manner and pitfalls early in the game can be avoided. Furthermore, moves played from an opening library do not have to be searched. This results in a time savings which can be applied to later moves. But an opening book is not a panacea; there are some possible problems. Accuracy is vital. Since the computer plays the move without question or examination, an error in the book can spell disaster. Assuming that error checking is adequate, there is still the problem that the computer may not understand the position that results from the master's moves. For example, the program may have pushed a Knight Pawn while in book, but may not fianchetto the Bishop if it falls out of book. Or the computer may "dislike" the piece arrangement that results and frantically reshuffle the pieces to suit its evaluation function. Pawn sacrifices in book are particularly prone to problems. The computer recipient of a sacrificed pawn may struggle to retain the material at a heavy cost to its position, or the computer donor of a pawn may seek only to regain the lost pawn while ignoring the attacking chances the sacrifice afforded. All in all, the advantages outweigh the disadvantages. Thus for most computer chess programs, the question is not whether to include an opening book, but how to encode it.

The single most crucial factor in encoding opening moves is space. Time is not critical, since the move need only to be located and played. Space is important because the more compact the representation, the more positions can be stored in the allotted area, and, in opening books, "the more the merrier." Several representations are possible. Perhaps the simplest is to encode the "from-square" of the moving piece and the "to-square" of its destination. Although simple, this technique uses a fair amount of memory (6 bits per square times 2 squares). Another possible representation is hashing. Hashing has the advantage that positions can be located directly in the book without reference to the moves that produced the position. Thus hashing facilitates location of transpositions. The human opponent will not find it so easy to throw the computer out of book by altering the sequence in which the moves are played. If a book position results, the hashed scheme will find it despite the transposed moves. A hashing scheme does not necessarily describe the position exactly. It is possible for different positions to hash to the same location. If this happens, a collision results. Collisions can be as disastrous as typing errors to an opening book. Furthermore, hashing can be consumptive of memory as well, and is essentially a technique for the main frames. Another possible move representation technique is a move descriptor. A data field (one byte or larger) is used to describe the move (e.g. the White Knight, nearest to the lower left-hand corner of the board, moves up and to the right two squares). It is possible to generate a descriptive system that uses only eight bits per move, so it has the advantage of compactness. The disadvantage is mainly the complexity of the opening book and how hard it is to generate the book data.

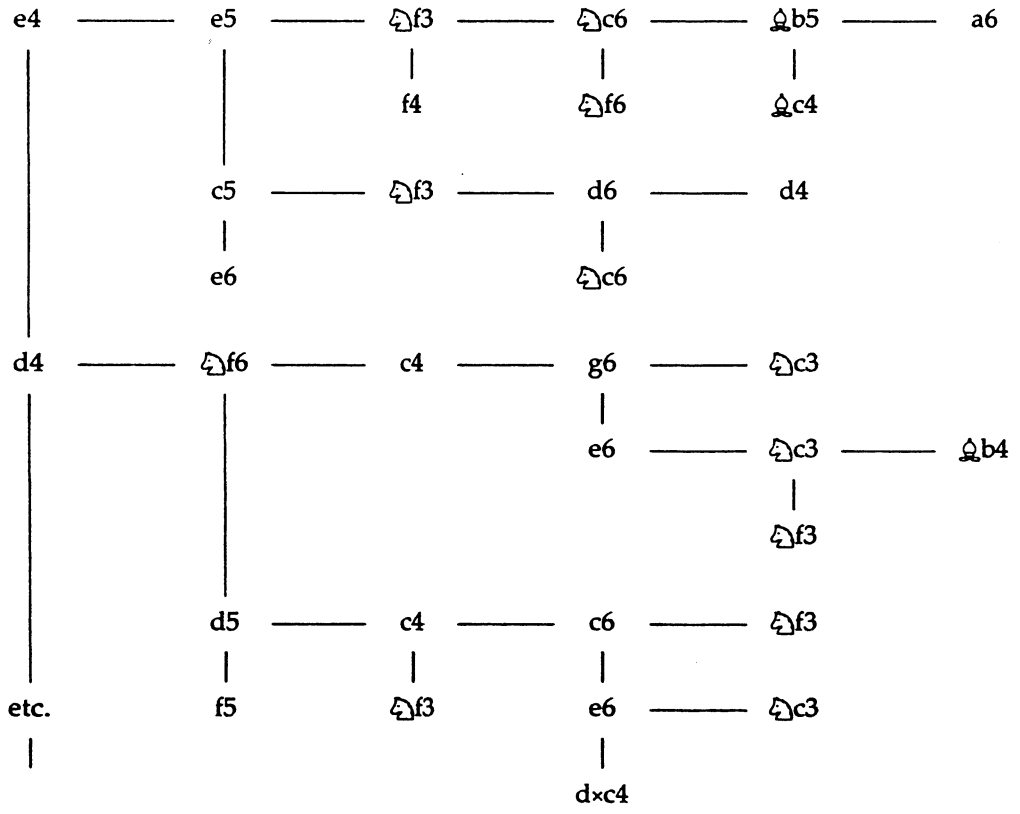
One of the simplest and most compact forms of book move representation is to use a "displacement into the move generator." Every chess program must have a means of generating all legal moves in a given position. If the move generator is guaranteed to generate

the moves in the same order every time it is called in the same position, then it is possible to specify a book move as the *n*th move generated.

e.g. Suppose the move generator for a chess program generates pawn moves in the initial position in the following order: a3, a4, b3, b4, c3 and so on. Then the opening pawn move e4 can be specified as the 10th move generated.

The major disadvantage of this system is that it is position dependent. You must know where you are in the book to know whether the move is relevant. The advantages are simplicity and compactness, since the book moves can be encoded in 6 bits. If an 8-bit byte is taken as the basic unit, this leaves 2 bits per move and, as we shall see, this is data enough to describe the entire tree structure!

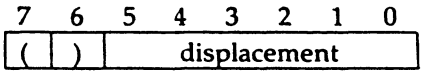
A move is of no value in isolation, but only in the context of an opening system. Many such systems have been explored in the chess literature. There are several choices of the first move and for each such first move, there are several possible responses. If transpositions and repeated positions are ignored for the moment, the opening system takes on the structure of a tree. For example:



Since a single move can lead to a host of responses, there is a need to define which moves go together. Conceptually simplest would be to duplicate the precursor moves. For example, in every sub-variation beginning with 1. e4 e5 2. ♘f3 ♘c6, these same moves would be repeated. Thus each possible path through the tree would be separately encoded. Such a technique is highly consumptive of memory and greatly discourages storing multiple variations. Another possible encoding scheme uses pointers to locate all the descendents of a given move. This system doesn't use as much memory as duplicating the moves, but costs the space for the pointers.

The method that I am about to explain was originally described to me by Ken

Thompson. It uses two bits to characterize the entire tree.



One bit, designated '(', indicates that there is another move choice available in this position. The other bit, designated ')', indicates that we have reached the end of that line. Thus the tree illustrated above can be described as:

(e4 (e5 (♘f3 (♘c6 (♗b5 a6)
♗c4)
♘f6)
f4)
(c5 ♘f3 (d6 d4)
♘c6)
e6)
(d4 (♘f6 c4 (g6 ♘c3)
e6 (♘c3 ♗b4)
♘f3)
(d5 (c4 (c6 ♘f3)
(e6 ♘c3)
d×c4)
♘f3)
f5)
etc.

Stored in memory, the tree would look like this:

(e4 (e5 (♘f3 (♘c6 (♗b5 a6) ♗c4) ♘f6) f4) (c5 ♘f3 (d6 d4) ♘c6) e6) (d4 (♘f6 c4 (g6 ♘c3) e6 (♘c3 ♗b4) ♘f3) (d5 (c4 (c6 ♘f3) (e6 ♘c3) d×c4) ♘f3) f5) etc.

Thus a tree containing *n* nodes can be represented in *n* bytes.

Now we have a scheme to store an opening repertoire very compactly. Is that all? How are choice-points handled for the computer? In Thompson's implementation, the computer only plays one possible move (that listed first) in any position. What if it is desired to introduce randomness into the book? The '()' is presently undefined. It can be used as an extension to indicate that the following 6 bits contain, not an encoded move, but a code that describes another function.

For example, bit code "11000011" might be used to indicate that the computer should recognize, but not play, the following move. While code "11000100" might mean that the position already reached is a transposition of moves elsewhere in the book. In that case the next byte or two bytes might contain a pointer to another location in the book. Transpositions can also be handled by "walking" the tree to see if the given position is present somewhere in the book. With a minimum of optimization to avoid examining lines that cannot possibly lead to the given position, the time cost for walking the tree is quite reasonable.

Other uses for the '()' code can be imagined. Move choices can be identified by style of game they lead to. e.g. open, or closed, or whether a gambit is offered. Such a system could allow the user to select a game suited to his mood.

All of these encoding schemes serve one goal — conserving space to allow a maximum number of positions to be represented. They do nothing to solve some of the problems of opening books. At a minimum, some system of testing must be devised. This testing can make sure the intended move is legal and can spot "give-away" goofs. But, for opening books to really achieve their potential, it is necessary to introduce some understanding of the positions reached. This represents a far greater challenge.